Interactive and Deterministic Data Cleaning

A Tossed Stone Raises a Thousand Ripples

Jian He[1]* Enzo Veltri[2] Donatello Santoro[2] Guoliang Li[1]
Giansalvatore Mecca[2] Paolo Papotti[3]* Nan Tang[4]

[1]Tsinghua University, China [2]Università della Basilicata, Potenza, Italy [3]Arizona State University, USA
[4]Qatar Computing Research Institute, HBKU, Qatar

{hej13, liguoliang}@tsinghua.edu.cn, ppapotti@asu.edu, ntang@qf.org.qa
{enzo.veltri, donatello.santoro, giansalvatore.mecca}@gmail.com

ABSTRACT

We present FALCON, an *interactive, deterministic, and declarative* data cleaning system, which uses SQL update queries as the language to repair data. FALCON does not rely on the existence of a set of pre-defined data quality rules. On the contrary, it encourages users to explore the data, identify possible problems, and make updates to fix them. Bootstrapped by one user update, FALCON guesses a set of possible SQL update queries that can be used to repair the data. The main technical challenge addressed in this paper consists in finding a set of SQL update queries that is minimal in size and at the same time fixes the largest number of errors in the data. We formalize this problem as a search in a lattice-shaped space. To guarantee that the chosen updates are semantically correct, FALCON navigates the lattice by interacting with users to gradually validate the set of SQL update queries. Besides using traditional one-hop based traverse algorithms (*e.g.,* BFS or DFS), we describe novel multi-hop search algorithms such that FALCON can dive over the lattice and conduct the search efficiently. Our novel search strategy is coupled with a number of optimization techniques to further prune the search space and efficiently maintain the lattice. We have conducted extensive experiments using both real-world and synthetic datasets to show that FALCON can effectively communicate with users in data repairing.

CCS Concepts

•Information systems → Extraction, transformation and loading; Data cleaning;

Keywords

Data Cleaning; Interactive; Deterministic; Declarative

*Work partially done while interning/working at QCRI.

SIGMOD '16, June 26–July 1, 2016, San Francisco, CA, USA.

DOI: http://dx.doi.org/10.1145/2882903.2915242

	Date	Molecule	Laboratory	Quantity
t_1	11 Nov	$C_{16}H_{16}Cl$	Austin	200
t_2	12 Nov	statin→$C_{22}H_{28}F$	Austin	200
t_3	12 Nov	$C_{24}H_{75}S_6$	N.Y.→ New York	1000→100
t_4	12 Nov	statin	Boston	200
t_5	13 Nov	statin	Austin	200
t_6	15 Nov	$C_{17}H_{20}N$	Dubai	150

Table 1: Dataset T_{drug} with drug tests.

1. INTRODUCTION

High quality data is important to all businesses, and data cleaning is an important but tedious step. In fact, removing errors in order to get high quality data takes most of data analysts' time [31], and some studies predict a shortage of people with the skills and the know-how for these tasks [33].

Consequently, the number and variety of users who are getting close to the data for data quality tasks are destined to increase, and we cannot assume that only IT staff and data scientists are in charge of the data cleaning process.

The above requirement poses new and interesting research challenges. Indeed, a large body of the research has been conducted on *rule-based data repairing*, which consists of using integrity constraints to identify data errors [11, 12, 17, 25, 40], and automated algorithms to enforce these constraints over the data [7, 22, 23, 32, 43]. However, in the evolving scenario of data cleaning, these approaches show a serious limitation. Specifically, they assume that data quality rules are declared upfront by domain experts who understand the data and write logical formulas or procedural code. Despite many promising results, these systems have failed short in terms of adoption in industrial tools.

We address the problem of improving the data cleaning process by involving non-expert users as first-class citizens, and present FALCON, a novel system for *interactive* data repairing. FALCON departs from other interactive data cleaning systems [20, 27, 37, 41, 46], since it brings together a simple, user-oriented interaction paradigm with the benefits of a declarative, *deterministic*, and expressive data quality language – SQL update (SQLU) queries. In fact, the system is bootstrapped by an update to the data made by the user to rectify an error; based on that, it infers a set of SQLU queries that can be used as data quality rules to correct more errors. We illustrate by example how it works.

Example 1: Table 1 reports a sample real-world dataset T_{drug} for experiments collected from different labs. Each record represents the quantity and date of a test done in

a lab over a certain molecule. Errors are highlighted. Consider the following three user updates.

Δ_1: t_3[Laboratory] ← "New York" (from "N.Y.")
Δ_2: t_3[Quantity] ← 100 (from 1000)
Δ_3: t_2[Molecule] ← "$C_{22}H_{28}F$" (from "statin")

There exist multiple interpretations for each update. For instance, two possible semantics behind Δ_1 could be either reformatting all "N.Y." to "New York" as shown in Q_1, or changing all Laboratory values to "New York" as shown in Q'_1, regardless of their original values.

Q_1: UPDATE T_{drug} SET Laboratory = "New York"
WHERE Laboratory = "N.Y.";

Q'_1: UPDATE T_{drug} SET Laboratory = "New York";

Similarly, one possible interpretation of Δ_2, as given in Q_2, is that it is specific for Molecule and Date. Hence, it is hard to generalize this update to apply it to other tuples.

Q_2: UPDATE T_{drug} SET Quantity = 100
WHERE Molecule = "$C_{24}H_{75}S_6$" AND Date = "12 Nov";

Update Δ_3 is more interesting. Consider the following three interpretations with different effects. Q_3 repairs errors in both t_2 and t_5. Q'_3 also repairs both t_2 and t_5, but additionally, it modifies t_4[Molecule] to "$C_{22}H_{28}F$", which is an erroneous update, since in Boston they test a different statin molecule. On the other hand, the tuple-specific query Q''_3 only corrects t_2 but misses the chance to repair t_5.

Q_3: UPDATE T_{drug} SET Molecule = "$C_{22}H_{28}F$"
WHERE Molecule = "statin" AND Laboratory = "Austin";

Q'_3: UPDATE T_{drug} SET Molecule = "$C_{22}H_{28}F$"
WHERE Molecule = "statin";

Q''_3: UPDATE T_{drug} SET Molecule = "$C_{22}H_{28}F$"
WHERE Molecule = "statin" AND Laboratory = "Austin"
AND Date = "12 Nov" AND Quantity = 200;

From Example 1, one may observe that there might exist a large number of SQLU queries. Indeed, this large number is not surprising, as up to thousands of precise and reliable update queries can be needed in real-world settings, such as Walmart catalog [14]. However, while an update is a perfect starting point for the process of inferring the general scripts, it comes with new challenges in terms of user interactions.

First, the search space for a new update is exponential to the number of the attributes, and domain experts cannot manually validate each of these SQLU queries. We have to assume that a *budget* (*e.g.,* #-user interactions) is given for a specific update. Second, the discovery algorithm must be fast (*e.g.,* able to react in seconds) to enable user interactions. However, each interaction may trigger the update of data, which makes the search space a *dynamic* environment. This *dynamic* behavior, together with the large search space and a budget of user capacity, prevents the use of traditional tools for interactive response, such as precomputing and caching. In order to efficiently manage all potential updates, and effectively interact with users, we propose FALCON, which works as follows.

Workflow. The workflow of FALCON is depicted in Figure 1. ❶ The user examines the data and provides a repair Δ over table T. ❷ Given Δ, FALCON generates a set of SQLU queries as rules. It then selects a query Q whose validity is yet unknown, and asks the user to verify it. ❸ Based on the user verification on Q to be either True (*i.e.,* valid) or False (*i.e.,* invalid), if Q is True, it utilizes Q to repair more data.

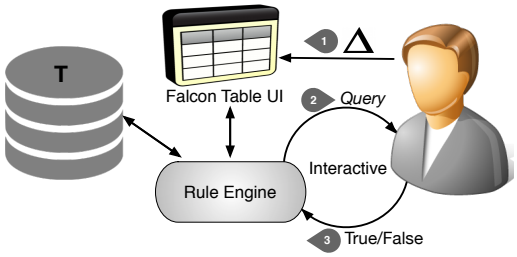


Figure 1: FALCON workflow.

Obviously, FALCON can prune the search space based on the validation on Q. The loop for steps ❷ and ❸ terminates when either all usable queries have been identified, or the user has no more capacity for the current Δ. Afterwards, the user may go back to step ❶ to inspect another repair.

Contributions. We present FALCON, a novel interactive data cleaning system, with the following contributions.

(1) To design data quality rules, we adopt the standard and deterministic language of SQL update statements (Section 2). We discuss how to organize the search space of candidate rules as a lattice, and its pruning principles, by leveraging the properties of the lattice (Section 3).

(2) We devise efficient algorithms for selecting candidate queries to effectively interact with the user (Section 4). In particular, in contrast to traditional traversal (one-hop) based approaches (Section 4.1), we present novel multiple-hop search algorithms such that FALCON can accurately discover useful queries in a small number of steps (Section 4.2).

(3) We describe optimization techniques to improve the efficiency of lattice maintenance (Section 5.1). We also propose *closed query sets* to compress the lattice so as to improve the search efficiency (Section 5.2).

(4) Implemented on top of an open-source data wrangling tool OpenRefine (http://openrefine.org), we have conducted experiments with real-world and synthetic data to show the effectiveness and efficiency of FALCON (Section 6).

Section 7 presents related work. Section 8 closes this paper, followed by our agenda for future work.

2. PROBLEM STATEMENT

We first introduce the rules used to repair data (Section 2.1). We then describe the search space of rules given one user update (Section 2.2) and formally define the problem studied in this paper (Section 2.3). Finally, we discuss its associated fundamental problems (Section 2.4).

2.1 SQL Update Queries: Mother Tongue

We adopt a simple and standard language to repair the database, the language of update statements in SQL (SQLU).

An SQLU statement updates records in a table T on attributes $A, B \ldots$, when some conditions hold. In this work, we restrict the language to the case where updates are done on one attribute A of table T with only boolean conjunctions:

UPDATE T SET $A = a$ WHERE boolean conjunctions

More specifically, each boolean conjunction is of the form $B = v_B$, where B is an attribute of table T and v_B is a constant value from the domain of B, *e.g.,* Molecule = "statin". Attribute B could also be the attribute to be updated (*i.e.,* $B = A$), such as Laboratory in Q_1 of Example 1.

We shall use the terms SQLU queries and *data quality rules* (or simply *rules*) interchangeably in the following. We will also treat *updates* and *repairs* equally.

Remark. SQLU queries used in this work are quite different from the integrity constraints (ICs) that are widely adopted by other data cleaning systems, such as functional dependencies [1], conditional functional dependencies [16], conditional inclusion dependencies [7], and denial constraints [12]. ICs are used to capture errors as violations, where one violation is a set of values that is not semantically coherent when putting together. In other words, ICs do not explicitly specify how to change data values to resolve violations. In contrast, SQLU statements explicitly specify how to change data values, which are thus considered to be *deterministic*. The proposed SQLU is powerful enough to support existing deterministic cleaning languages such as fixing rules [43], constant CFDs [16], and widely used ETL rules.

Note that in this work we restrict our discussion to conjunctive SQLU queries for three reasons. (1) It is easy for users to understand, which is important for interacting with users; (2) It is efficient to reason about the relationship between different queries; and (3) It is known that queries with other formulae such as disjunctions or negations can be rewritten into an equivalent conjunctive formula [1].

2.2 Search Space for One Repair

Consider a repair $\Delta : t[A] \leftarrow a'$ that changes the value of $t[A]$ from error a to its correct value a' with $a \neq a'$. We want to generalize this action so as to repair more errors.

Naturally, there exist multiple queries to interpret this repair Δ. Implicitly, for each query, the SET clause is $A \leftarrow a'$. Hence we focus on the WHERE clause. Consider a boolean condition as $B = v_B$, where B could be any attribute in relation R. In an *open-world* assumption, the constant v_B can be assigned from an infinite set of values, which is neither reasonable nor feasible in practice. Instead, we adopt a *closed-world* assumption by only using the evidence from tuple t, the tuple that is being repaired. In other words, for a query Q *w.r.t.* the above update Δ, if an attribute B appears in the WHERE condition of Q, then the boolean conjunction is $B = t[B]$, which is to bind the constant v_B to the value $t[B]$. As a special query, we consider $\varnothing$ as *no condition* being enforced in the WHERE clause. Stating in another way, it is to update all A values in T to a'.

In summary, given a repair $t[A] \leftarrow a'$ for tuple t in table T of relation R, the set $\mathcal{Q}$ of all rules for such a repair is:

UPDATE $\boxed{T}$ SET $\boxed{A = a'}$ WHERE $\boxed{X = t[X]}$

where X is an arbitrary subset of R, which can range from the empty set $\varnothing$ to all attributes in R (*i.e.,* $X = R$). Hence, there are $2^{|R|}$ possibilities of X, where $|R|$ is the arity of relation R. In other words, we can infer $2^{|R|}$ queries for each update. Consider update Δ_3 in Example 1, we can infer $2^4 = 16$ queries, where three of them are shown as Q_3, Q_3' and Q_3''.

2.3 Problem Statement

Given a repair, one wants to find the queries that are semantically correct so as to repair the database.

Valid SQLU query. Given a repair, an SQLU query is *valid* if the query is semantically correct. Since we do not know which queries are valid in advance, we need to ask the user to either validate the query as semantically correct, or invalidate it otherwise. Naturally, we want to find all valid SQLU queries and use them to repair the database. A straightforward strategy is to ask the user to check every possible query. Of course, this method is rather expensive as there could be a large number of possible queries, for which we will use containment relationships among queries to improve the search of queries (Section 3).

Furthermore, the user normally has limited capacity for the number of queries he/she can verify. To this end, we want to find the *cost-effective queries* to maximize the number of repaired tuples based on the queries validated by the user, which is formally defined below.

Budget repair problem. Given a set $\mathcal{Q}$ of SQLU queries, a table T, and a budget B for the number of interactions the user can afford, the *budget repair problem* is to select B queries $\mathcal{Q}'$ from $\mathcal{Q}$, so as to maximize $|\bigcup_{Q \in \mathcal{Q}' \wedge \mathsf{valid}(Q)=\mathsf{T}} Q(T)|$.

Here, $\mathsf{valid}(Q)$ is a boolean function that is T (resp. F) if Q is a valid query (resp. not), and $Q(T)$ represents the set of repairs of applying query Q over table T.

Observe that in the above problem, given a query Q, the validity of Q (*i.e.,* $\mathsf{valid}(Q)$) is unknown, to be verified by the user. Such a problem is typically categorized under the framework of *online algorithms* [3], where one can process input piece-by-piece in a serial fashion (*i.e.,* the verification $\mathsf{valid}(Q)$ of some Q), without having the entire input (*i.e.,* the value $\mathsf{valid}(Q)$ for each Q in $\mathcal{Q}$) available from the start.

Offline problem. Its corresponding *offline variant* is the following. Given as input that whether each query Q in $\mathcal{Q}$ is valid or not is known, how to select B queries from $\mathcal{Q}$ to maximize the number of repaired tuples. The objective of designing an online algorithm is to get answers as accurate as the offline problem. It is easy to see that the offline problem of its online version (*i.e.,* the budget repair problem) is NP-hard, which can be readily proved by a reduction from the maximum-coverage problem [34].

On analogy of what is proved in [5], when the offline variant is NP-hard, there is no efficient algorithm for computing an optimal solution for its online algorithm. In other words, when the offline variant is intractable, there is no hope to find an optimal solution with the cost in a constant factor of the online variant (*a.k.a.* a *competitive analysis* [39]).

However, not all is lost. As will be shown later, we can organize all queries in a graphical structure, such that when the user verifies a query Q as valid or invalid, we can even generate more inputs by computing the validity of queries Q' that are related to Q (Section 3). Even better, we devise efficient algorithms to search over the above graphical structure (Section 4) and empirically show the effectiveness of the presented strategies (Section 6).

2.4 Fundamental Problems

Let $\mathcal{Q}^+$ be a set of valid queries *w.r.t.* one user update.

Termination problem. The *termination problem* determines whether a rule-based process will stop, given $\mathcal{Q}^+$ and an instance T. We can readily verify that no matter in what order the queries in $\mathcal{Q}^+$ are executed, the whole process will terminate, since the execution of each query is deterministic.

Conflicting queries. Two queries Q_1 and Q_2 are *conflicting queries* if there exists a tuple t' such that the following

two sequences of SQL updates will obtain different results: (1) $Q_1(Q_2(t'))$, *i.e.,* applying Q_2 first to t followed by Q_1, and (2) $Q_2(Q_1(t'))$.

Note that, the search space *w.r.t.* one repair $\Delta : t[A] \leftarrow a'$ is a set $\mathcal{Q}$ of queries (Section 2.2), where each query $Q \in \mathcal{Q}$ is a way to generalize the action of changing $t[A]$ to a specific value a', by considering different attribute combinations. In other words, no query Q will change a tuple to a value a'' that is different from a'. Hence, conflicting queries will not be generated in one lattice.

Determinism problem. The *determinism problem* asks whether all repairing processes (with different repairing orders of the SQLU queries) end up with the same repair, given $\mathcal{Q}^+$ and an instance T.

It is easy to verify that, given $\mathcal{Q}^+$ and T, regardless of the orders of the queries in $\mathcal{Q}^+$ are applied, all data repairs are $\bigcup_{Q \in \mathcal{Q}^+} Q(T)$, where T is the original instance. Hence, any set of rules is trivially deterministic.

3. A LATTICE: FALCON SEARCH SPACE

In this section, we shall present our organization of the search space, so as to enable both efficient and effective search over the candidate rules. We start by discussing the relationship between two data quality rules.

Rule containment. For two rules Q and Q', we say that Q is *contained* by Q' (or Q' *contains* Q), denoted by $Q \preceq Q'$, if for all possible database instances T over the input schema R, the result of $Q(T)$ is a subset of the result of $Q'(T)$ (*i.e.,* $Q(T) \subseteq Q'(T)$).

Intuitively, the rule containment captures the semantic relationship among rules. In other words, no matter which database T is used, Q will update a subset of T tuples that Q' will update if $Q \preceq Q'$, since Q is more specific than Q'.

Example 2: Consider queries Q_3, Q_3' and Q_3'' in Example 1. It is straightforward to see that both Q_3 and Q_3'' are contained by Q_3' (*i.e.,* $Q_3 \preceq Q_3'$ and $Q_3'' \preceq Q_3'$), and Q_3'' is contained by Q_3 (*i.e.,* $Q_3'' \preceq Q_3$).

It is readily to verify that the query containment "$\preceq$" is a partial order over the set $\mathcal{Q}$ of all possible rules, which is reflexive, antisymmetric, and transitive. More specifically:

[**Reflexivity**] $Q \preceq Q$, for any $Q \in \mathcal{Q}$.
[**Antisymmetry**] If $Q \preceq Q'$ and $Q' \preceq Q$, then $Q = Q'$.
[**Transitivity**] If $Q \preceq Q'$ and $Q' \preceq Q''$, then $Q \preceq Q''$.

For a query Q, we denote by $\mathsf{attr}(Q)$ the set of distinct attributes in its WHERE condition.

Note that for each user update, the SQLU queries have the same value constraint on the same attribute, and thus the rule containment verification is *equivalent* to a simpler condition: $Q \preceq Q'$ if $\mathsf{attr}(Q')$ is a subset $\mathsf{attr}(Q)$. For instance, $Q_3 \preceq Q_3'$ since $\mathsf{attr}(Q_3') = \{\mathsf{Molecule}\} \subseteq \{\mathsf{Molecule}, \mathsf{Laboratory}\} = \mathsf{attr}(Q_3)$.

Affected tuples. For each query Q and instance T, we call the tuples in $Q(T)$ *affected tuples*, *i.e.,* the tuples that Q will repair. We also call $|Q(T)|$ the *affected number* of Q, relative to T. Consider Q_3 and T_{drug} in Example 1 for instance. The affected tuples are $Q_3(T_{\mathsf{drug}}) = \{t_2, t_5\}$, and its corresponding affected number is $|Q_3(T_{\mathsf{drug}})| = 2$.

We discuss next how to organize these queries to facilitate search strategies.

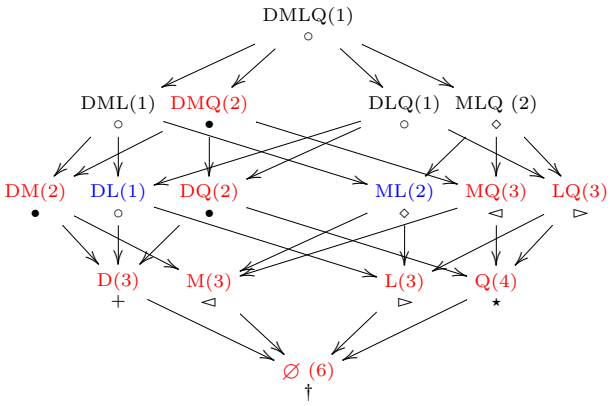


Figure 2: A sample lattice graph.

A set with a partial order is a *partially ordered set*, or *poset*. Hence, $\mathcal{Q}$ is a *poset* on the partial order $\preceq$ of rule containment. Moreover, consider any two rules Q and Q'. They have a *greatest lower bound*: the most specific query that is contained by both Q and Q'. This query, denoted by $Q \wedge Q'$, is the one *w.r.t.* $\mathsf{attr}(Q) \cup \mathsf{attr}(Q')$. Also, they have a *least upper bound*: the most general query that contains both Q and Q'. This query, denoted by $Q \vee Q'$, is the one *w.r.t.* $\mathsf{attr}(Q) \cap \mathsf{attr}(Q')$. Therefore, we can organize the queries in our search space as a *lattice*.

Query lattice. Given a repair Δ and a database instance T, we denote by $(\mathcal{Q}, \preceq, T)$ the corresponding lattice, or simply $(\mathcal{Q}, \preceq)$ when T is clear from the context. Each node in the lattice corresponds to a query $Q \in \mathcal{Q}$. Each directed edge from node Q to Q' indicates that $Q \preceq Q'$ (Q is contained in Q') and $|\mathsf{attr}(Q)| = |\mathsf{attr}(Q')|+1$ (with one different attribute). Moreover, the affected number associated with each query is maintained in the lattice (we will discuss how to compute the number in Section 5.1.2).

Example 3: Figure 2 depicts the lattice for dataset T_{drug} and update Δ_3 given in Example 1. Each capital letter is an abbreviation of an attribute, *e.g.,* D for Date. The node ML is for the query Q_3 on attributes Molecule and Laboratory. The edge from ML to M indicates that the query Q_3 (for ML) is contained in Q_3' (for M). The number 2 in node ML is the affected number of $|Q_3(T_{\mathsf{drug}})|$. Moreover, the greatest lower bound (resp. lowest upper bound) of ML and DL is MDL (resp. L). We postpone the discussion of the shapes in the figure, *e.g.,* "$\triangleright$", "$\star$" and "$\circ$", to Section 5.2.

Valid and maximal valid nodes. Given a lattice $(\mathcal{Q}, \preceq)$, the node relative to a rule Q is *valid* if it is semantically correct, thus should be executed to repair data. In our work, if the validity of a rule is unknown, we rely on the user to verify (see more details in Section 2.3). Fortunately, if a rule Q is known to be valid, we can infer that Q' is also valid if $Q' \preceq Q$. Moreover, the node relative to a valid rule Q is *maximal valid*, if no Q'' is valid and $Q \preceq Q''$.

Example 4: Consider the lattice in Figure 2. Assume that there are two valid queries to be applied: ML (Q_3 in Example 1); and the other query DL that represents on a certain date a certain lab works on only one molecule. All red nodes are invalid queries, *i.e.,* the queries that users will semantically invalidate. The other nodes are valid nodes. Moreover, the blue nodes DL and ML are maximal valid nodes.

One nice property of using a lattice is that it provides opportunities to prune nodes to be visited during traversal.

Lattice pruning. If a node Q is valid, by inference, all nodes Q' where $Q' \preceq Q$ are valid. On the other hand, if a node Q is invalid, by inference, all nodes Q'' where $Q \preceq Q''$ are invalid. The rationality behind the above inferences is that: if one query is valid, then any query that is more specific is also valid; conversely, if it is invalid, then any query that is more general is also invalid.

We denote by $Q^{\Uparrow}$ (*i.e.,* above Q in the lattice) the queries that Q contains, and $Q_{\Downarrow}$ (*i.e.,* below Q in the lattice) the queries that contain Q. These notations naturally extend to a set of queries, $\mathcal{Q}^{\Uparrow}$ and $\mathcal{Q}_{\Downarrow}$, such that $\mathcal{Q}^{\Uparrow} = \bigcup_{Q \in \mathcal{Q}} Q^{\Uparrow}$ and $\mathcal{Q}_{\Downarrow} = \bigcup_{Q \in \mathcal{Q}} Q_{\Downarrow}$.

Example 5: Consider again the lattice in Figure 2. During interactions with the user, if DL is validated, we can then derive that $\text{DL}^{\Uparrow} = \{\text{DML, DLQ, DMLQ}\}$ is valid. Consider now DQ, if DQ is invalidated, we can then derive that $\text{DQ}_{\Downarrow} = \{\text{D, Q, } \varnothing\}$ is invalid.

The notation used in this paper is summarized in Table 3 in Appendix A.

4. ALGORITHMS: FALCON IN ACTION

In this section, we first describe some traversal based algorithms to solve our budget repair problem (Section 4.1). We then present advanced algorithms to efficiently navigate the search space (Section 4.2).

When discussing the algorithms, we assume that the lattice has been built given the user provided repair. The algorithms are designed for traversing the lattice and interacting with the user. Details of constructing and maintaining the lattice will be provided in Section 5.1.

4.1 One-Hop Search: Falcon Glide

In traditional traversal algorithms of a lattice $\mathcal{L}$ the search is based on some seeds, and then neighbours of the seeds (*i.e.,* one-hop) are visited by following edge connections. For example, Breadth-first search (BFS) traverses $\mathcal{L}$, by starting at the bottom and explores the neighbor nodes first, before moving to the next level neighbors. Depth-first search (DFS) differentiates in that after visiting a node, it explores as far as possible along each branch before backtracking. A recent traversal proposal, Ducc [28], bootstraps the search with a DFS-style exploration until a node of interest is found. Then it traverses the lattice alternating visits over valid and invalid nodes, in order to identify the border between them. While the algorithm was defined to find minimal unique column combination, it can be used for any lattice traversal.

To better understand how different algorithms work, we illustrate by an example below.

Example 6: Figure 3 shows how various search algorithms work, where red nodes indicate invalid nodes, blue nodes represent maximal valid nodes, and the other nodes (*i.e.,* small circles) are valid nodes. Let $B = 3$, the number of questions the user can verify. BFS search will visit the nodes in a breadth-first fashion, *e.g.,* in the order $B1, B2, B3$. DFS search will visit the nodes in a depth-first fashion, *e.g.,* in the order $D1, D2, D3$. Different from BFS and DFS, Ducc [28] explores the graph in a zigzag fashion, which tries to pivot on valid nodes and explores their neighbors, *e.g.,* in the order $A1, A2, A3$. Since the above methods are edge based, the search paths are indicated on edges.

Now let us give some insight why traversal based algorithms fail for our problem. Nodes close to the top (resp. bottom) are more likely to be valid (resp. invalid). Hence, if we traverse the lattice *top-down*, we have more chances to visit a valid node Q. However, since it is close to the top, the number of inferred valid nodes $Q^{\Uparrow}$ is small. On the other hand, if we traverse the lattice *bottom-up*, we have more chances to visit an invalid node Q'. However, since it is close to the bottom, the number of inferred invalid nodes $Q'_{\Downarrow}$ is small.

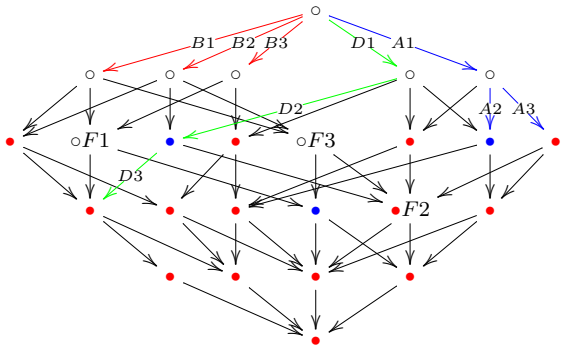


Figure 3: Lattice search algorithms. (Nodes ○/●/● represent valid nodes/maximal valid nodes/invalid nodes. Red/green/blues edges are used to explain different search strategies: BFS/DFS/Ducc.)

As shown in Example 6 and the above discussion, traversal based algorithms are locality based – they follow edge connections from visited nodes. In such a way, FALCON can only *glide* over the lattice. This is obviously not ideal when the lattice is big but the budget B is small, which is exactly the case we face. Hence, we propose new algorithms next.

4.2 Multi-Hop Search: Falcon Dive

Now that we know that traversal based algorithms are not suitable for our studied problem, we need to devise new algorithms so that FALCON can *dive* on the lattice.

4.2.1 Binary Jump

Given a budget B, our objective is to define a *divide-and-conquer* strategy that efficiently identifies nodes that are both valid and not very close to the top, so as to maximize the number of tuples to be repaired. To this purpose, we present an strategy, namely *binary jump*, inspired by classical binary search. Roughly speaking, we treat the search space as a linear space (*i.e.,* an array) by sacrificing some structural connections, and sort the nodes based on their associated affected numbers. We can then do multi-hop search to locate a candidate node to be verified with the user.

Note that conventionally, *a binary search* finds the position of a target value within a sorted array. Different from it, *binary jump* does not have a target value to be searched. In other words, binary jump is just inspired by binary search by doing half-interval style lattice traversal.

Binary jump over a path. We first discuss binary jump over a path. Consider DMLQ → DLQ → LQ → Q → ∅ in Figure 2. The ground truth of the validity of them is (T, T, F, F, F), where T means valid and F means invalid. The search algorithm does not know the ground truth, so initially we have (?, ?, ?, ?, ?). To find the truth with traversal based approaches, we need $O(N)$ questions in average, where N is the length of the path. However, using binary

jump will reduce it to $O(\log N)$ questions, which is optimal, by applying inferences of finding all valid/invalid nodes.

Next we discuss the meaning of "*binary*". Straightforwardly, *binary* may refer to the offset as standard binary search. However, we need to incorporate the information of affected number. Hence, the *binary* search could refer to the median number. For instance, in Figure 2, the path DMLQ $\rightarrow$ DLQ $\rightarrow$ LQ $\rightarrow$ Q $\rightarrow \varnothing$ corresponds to the affected numbers $(1, 2, 3, 4, 6)$ and the binary jump is to find the value that is closest to $\lceil (1+6)/2 \rceil = 4$.

For binary jump, we introduce a parameter d to bound the search depth, which is the number of iterations one can do binary jump before termination. Given a path $Q_1, Q_2, \cdots, Q_x$, we first ask the middle node $Q_{x/2}$. If the node is valid, we ask the next middle node between $Q_{x/2}$ and Q_x; otherwise, we ask the next middle node between Q_1 and $Q_{x/2}$. After d wrong searches, the process terminates. We refer to this search strategy as BinaryJump(). The rationale behind using the parameter d is that if we are following the wrong direction, we should be aware and go back to the right track, as *a fault confessed is half redressed*. We will discuss how d is set in practice in Section 6.

Note that the number of the most general query (*i.e.,* the empty set at the bottom of the lattice) will change the whole column, which makes the median number an optimistic estimation. To make it more realistic, instead, we set the binary jump using log scale to find the value that is closest to *e.g.,* $\lceil \log_2^{(1+6)} \rceil = 3$.

From a path to a lattice. In order to take the advantage of binary jump for lattice traversal, the broad intuition is to do dimension reduction from a lattice to a one-dimensional structure. That is, if we treat all nodes in the lattice uniformly, by sorting them in ascending order on their associated affected numbers, we get a sorted array similar to the one discussed above for the path.

Let $\mathcal{Q}^?$ denote a set of unvalidated nodes, $\mathcal{Q}^+$ represent a set of valid nodes, and $\mathcal{Q}^-$ indicate a set of invalid nodes. Next we present the algorithm.

Algorithm. Given a lattice $(\mathcal{Q}, \preceq)$ *w.r.t.* a repair Δ over table T, a budget B for the number of questions the user can answer, and a depth d to bound the search depth, the algorithm for binary jump is given below.

D1. [Initialization.] Let $\mathcal{Q}^- = \varnothing$, $\mathcal{Q}^+ = \{\mathsf{top}\}$ (the top node of the lattice), and $\mathcal{Q}^? = \mathcal{Q} \backslash (\mathcal{Q}^- \cup \mathcal{Q}^+)$. Also, let $\mathcal{Q}^\checkmark$ be the set of nodes verified by users, initially empty.

D2. [Sort.] Sort unvalidated nodes $\mathcal{Q}^?$ based on their affected numbers in ascending order.

D3. [Binary jump] Do the binary jump over $\mathcal{Q}^?$ and select one node Q, which is referred to as BinaryJump(). If the user still has capacity (the total number of interactions is below B), it interacts with user to verify Q, and updates $\mathcal{Q}^\checkmark = \mathcal{Q}^\checkmark \cup \{Q\}$. Otherwise, the whole process terminates. If Q is valid, it goes to step **D4**; otherwise, it goes to **D5** below, if Q is invalid.

D4. [Q is valid.] Apply Q over table T and update the affected numbers of nodes in $\mathcal{Q}$. Set $\mathcal{Q}^+ = \mathcal{Q}^+ \cup Q^{\uparrow\!\!\uparrow}$ (infer and enlarge valid nodes). Let $\mathcal{Q}^? = Q_{\perp\!\!\perp}$ and go to step **D2**.

D5. [Q is invalid.] Set $\mathcal{Q}^- = \mathcal{Q}^- \cup Q_{\perp\!\!\perp}$ (infer and enlarge invalid nodes). If the current depth is d, it goes to step **D6**. Otherwise, $\mathcal{Q}^? = Q^{\downarrow\!\!\downarrow}$ and goes to step **D2**.

D6. [New search space.] Let $\mathcal{Q}^? = \mathcal{Q} \backslash (\mathcal{Q}^{\checkmark\downarrow\!\!\downarrow} \cup \mathcal{Q}^{\checkmark}{}_{\perp\!\!\perp})$, *i.e.,* search on the nodes that are not linked to any verified node. It then goes to step **D2**.

Complexity. It is easy to see that there are up to B iterations, and the sort (**D2**) dominates the cost. Hence, the total time complexity is $O(B \cdot |\mathcal{Q}| \cdot log|\mathcal{Q}|)$. Here, budget B is typically small. Although the size of $\mathcal{Q}$ could be large for a big relation, we will discuss an optimization in Section 5.1.1 about how to ensure that the size of $\mathcal{Q}$ is easily manageable.

4.2.2 Attribute Correlation: A Good Bait

Intuitively, we want to greedily select at each step the node Q that is more likely to repair a large number of tuples. However, since we do not know what are the correct nodes until we verify them with the user, we need to estimate this information. To define the score of a node, we augment the existing information on the affected number of each query Q, *i.e.,* the number of A values Q can repair (Section 3), with the likelihood of a certain node to be related to the current attribute A.

Attribute correlations. The *attribute correlation* between two attributes A and B, denoted by $\mathsf{cor}(A, B)$, is to measure how close they are to each other.

The intuition of using attribute correlations is that, if node Q is correlated to attribute A that is being updated, then it is more likely to be semantically relevant and useful for the repair process. In general, we may get such information from data profiling tools that measure attributes correlation.

We adopt the techniques proposed in CORDS [29] to profile a database T of relation R. Specifically, CORDS computes for each attribute pair a score in $[0, 1]$. Note that (A, B) and (B, A) are different pairs. The score of an attribute pair (A, B) equals to 1 means that it is a soft FD, indicating that A approximately uniquely determines B. Otherwise, it is a score computed using χ^2 statistics by examining the attribute values in attributes A and B.

In our lattice, oftentimes, we want to estimate the correlation between the attributes in a query Q (*i.e.,* $\mathsf{attr}(Q)$) and the attribute A being updated. In other words, we need to compute the correlation between a set of attributes to a single attribute.

Using attribute correlations. We modified the algorithm presented in CORDS to compute the correlation between a set X of attributes and an attribute A, denoted by $\mathsf{cor}(X, A)$. In CORDS, an attribute pair (A, B) is a soft FD if the support value $\mathsf{sup}(A, B)$ is above a given threshold τ (see [29] for more details). Similarly, we output (X, B) as a soft FD if the support value $\mathsf{sup}(X, B) > \tau$. Otherwise, we compute the correlation score in $[0, 1]$ for (X, B) as follows.

$$\mathsf{cor}(X, B) = \frac{\chi^2}{nq} \tag{1}$$

$$\chi^2 = \sum_{v_1=1}^{m_1} \sum_{v_2=1}^{m_2} \cdots \sum_{v_k=1}^{m_k} \frac{(n_{v_1,v_2,\ldots,v_k} - e_{v_1,v_2,\ldots,v_k})^2}{e_{v_1,v_2,\ldots,v_k}} \tag{2}$$

$$e_{v_1,v_2,\ldots,v_k} = n \prod_{j=1}^{k} Pr(v_j) = n \prod_{j=1}^{k} \frac{n_{v_j}^j}{n} = \frac{n_{i_1}^j n_{v_2}^j \ldots n_{v_k}^j}{n^{k-1}} \tag{3}$$

$$q = \prod_{i=1}^{k} m_i - \sum_{i=1}^{k} m_i + k - 1 \tag{4}$$

Here, k is the number of attributes in X and m_i is the number of distinct values in the i-th attribute. Moreover, $(v_1, v_2, ..., v_k)$ is a tuple where the value of the j-th attribute is v_j. Also, $n_{v_1,v_2,\ldots,v_k}$ is the frequency of tuple

	Austin	N.Y.	Boston	Dubai	
$C_{16}H_{16}Cl$	1	0	0	0	1
statin	2	0	1	0	3
$C_{24}H_{75}S_6$	0	1	0	0	1
$C_{17}H_{20}N$	0	0	0	1	1
	3	1	1	1	

Table 2: A 2-way contingency table.

$(v_1, v_2, ..., v_k)$, and $e_{v_1,v_2,...,v_k}$ is the estimated frequency based on the probability of v_j appearing in the j-th attribute, *i.e.,* $n^j_{v_j}/n$, where $n^j_{v_j}$ is the frequency of v_j in the j-th attribute and n is the number of tuples.

Example 7: Consider Table 1, and a given soft FD in the traditional form: $\{\mathsf{Molecule}, \mathsf{Laboratory}\} \rightarrow \mathsf{Quantity}$. Naturally, we have that the correlation value for $\mathsf{cor}(\{\mathsf{Molecule}, \mathsf{Laboratory}\}, \mathsf{Quantity}) = 1$, since they can be verified from the soft FD given above.

Consider now $X = \{\mathsf{Molecule}\}$ and $B = \mathsf{Laboratory}$. Since there is no corresponding soft FD as $\{\mathsf{Molecule}\} \rightarrow \mathsf{Laboratory}$, we compute its correlation value by normalizing χ^2 statistics.

To do so, we first compute contingency table (see Table 2). We then compute expected count of each symbol tuple. Consider tuple {statin,Austin}. The expected count $e_{\mathsf{statin,Boston}} = (n^{\mathsf{Molecule}}_{\mathsf{statin}} \cdot n^{\mathsf{Laboratory}}_{\mathsf{Boston}})/n = 0.5$, and the real count $n_{\mathsf{statin,Austin}} = 1$. Thus the difference is $(n_{\mathsf{statin,Austin}} - e_{\mathsf{statin,Boston}})^2/e_{\mathsf{statin,Boston}} = 0.5$. By summing up all differences we have $\chi^2 = 12.67$, the degrees of freedom $q = 4 \cdot 4 - (4+4) + 2 - 1 = 9$, thus $\mathsf{cor}(\{\mathsf{Molecule}\}, \mathsf{Laboratory}) = 12.67/(6 * 9) = 0.235$.

We now give our greedy algorithm for multi-hop search driven by correlation and affected number.

Correlation aware binary jump (CoDive). We revise binary jump by using the correlation information, affecting **D3** in Section 4.2.1. Note that the function BINARYJUMP() will locate a node Q in the sorted list $\mathcal{Q}^?$. Instead of asking the user to verify Q, we revise it with the following methodology. (1) We pick more nodes around Q in the sorted list, with w on its left and the other w on its right. (2) For the above $2w+1$ nodes, we compute their scores (affected number multiplies correlation score) and select the one with the largest score, which will then be verified by the user. We will discuss how w is set in practice in Section 6.

5. OPTIMIZATIONS

In this section, we first discuss optimizations for maintaining the lattice (Section 5.1). We then describe a technique to compress the search space, which can be applied to all algorithms (Section 5.2). We also discuss an extension when external sources are available (Appendix B).

5.1 Lattice Maintenance

There are two main challenges when maintaining the lattice: its potential large size, and the updates of affected numbers of lattice nodes during each interaction. We address these two issues below.

5.1.1 Partial Lattice Materialization

For some dataset, the number of attributes in R can be large, such that a full materialization of the lattice is prohibitively expensive with $2^{|R|}$ nodes.

Fortunately, in our framework, the update provided by the user is a strong indicator to guide which attributes should be used. The intuition is that, given an update $t[A] \leftarrow a'$, not all attributes are relevant. Consequently, constructing a lattice by incorporating irrelevant attributes will decrease both efficiency and effectiveness. Hence, we propose to pick top-k attributes that are related to the attribute A being updated, based on the attribute correlation score discussed in Section 4.2.2. We refer to such a strategy as *partial lattice materalization*, which performs much faster than a full materialization of the entire lattice, without losing accuracy. This reduces the time complexity from $O(2^{|R|} \cdot |T|)$ to $O(2^k \cdot |T|)$ where k could be much smaller than $|R|$ in practice.

Practically, attribute correlation plays an important role in devising effective search strategies. We combine functional dependencies (FDs) and highly related attribute sets (rules) to improve the search strategy. Please see the experiment in Appendix D.1 for more details on this point.

5.1.2 Initialize and Maintain Affected Numbers

Initialization. Given an update $t[A] \leftarrow a'$, we need to compute the affected number of each query Q in the lattice. The straightforward way of executing an SQLU query for each node is very costly.

We approach the problem of *initializing affected numbers* by leveraging the containment relationships between nodes. Consider two queries Q and Q', if $Q \preceq Q'$, then given any database T, we have $Q(T) \subseteq Q'(T)$. Clearly, we can compute the result of $Q(T)$ from $Q'(T)$. This is exactly the problem of answering queries using materialized views [26]. Given the simplicity of the SQLU queries adopted in this work, the query rewriting is simply to apply a selection using a constant value.

Example 8: Consider two queries Q_3, Q'_3, and the dataset T_{drug} in Example 1. If we compute Q'_3 over T_{drug} first as $Q'_3(T_{\mathsf{drug}})$, the result of $Q_3(T_{\mathsf{drug}})$ is simply to select all tuples from $Q'_3(T_{\mathsf{drug}})$ whose Laboratory values are Austin.

The above example suggests a simple way of computing affected numbers of lattice nodes in a bottom-up fashion. Indeed, only one SQLU query is needed for the bottom node of the lattice. Afterwards, in the bottom-up procedure, for each query Q, it applies the aforementioned query rewriting technique on $Q'(T)$ to compute $Q(T)$, where $Q \preceq Q'$ indicates that Q' is one level below Q.

Maintenance. Given the lattice $(\mathcal{Q}, \preceq)$ for table T and update Δ, when some rule Q is validated by the user, the tuples affected by Q will be repaired, *i.e.,* $Q(T)$ will result in a repaired database T' where $T' = T \oplus Q(T)$, *i.e.,* applying Q to T. For each yet unvalidated rule Q', the above changes should be reflected, *i.e.,* the number of affected tuples should be changed correspondingly, from $|Q'(T)|$ to $|Q'(T')|$.

The straightforward way is to execute $Q'(T')$ to refresh $|Q'(T')|$, or an optimized way of using the query rewriting technique discussed above. However, in such incremental scenarios, incremental algorithms have been developed for various applications (see [36] for a survey). For incremental algorithms, the updates are typically computed from *affected areas*, not the entire dataset. In our case, the affect area is exactly the affected tuples $Q(T)$. Next, we discuss how to compute, for each unvalidated rule Q', the new $|Q'(T')|$.

Case 1 $[Q' \preceq Q]$**:** $|Q'(T')| = 0$.

Case 2 $[Q \preceq Q'']$**:** $|Q''(T')| = |Q''(T)| - |Q(T)|$.

Case 3 $[Q$ **and** Q''' **are disjoint]:** Neither $Q \preceq Q'''$ nor $Q''' \preceq Q$ holds. We have $|Q'''(T')| = |Q'''(T)| - |Q'''(Q(T))|$.

The above case 1 says that, if a valid rule Q is executed, then the tuples that can be affected by the queries Q' it contains have already been repaired. It is safe to set their affected numbers to 0 directly. The above case 2 tells that, for all the queries Q'' that contains Q, the set of tuples $Q(T)$ that Q'' can affect has been repaired. Hence, it is simple to reduce their affected numbers by $|Q(T)|$. In case 3, since neither $Q''' \preceq Q$ nor $Q \preceq Q'''$ holds, it first checks the number of tuples that Q''' can affect *w.r.t.* Q by executing $Q'''(Q(T))$, and then deducts its cardinality $|Q'''(Q(T))|$ from its maintained value $|Q'''(T)|$.

Time complexity. Cases 1 and 2 are clearly in constant time. For case 3, the cost is reduced from computing $Q'''(T')$ (*i.e.,* the entire table) to $Q'''(Q(T))$ (the tuples affected by Q) where $|Q(T)|$ is typically much smaller than $|T|$.

Example 9: Consider Fig. 2. Assume that during one interaction, the users validate ML (*i.e.,* query Q_3 in Example 1). The affected tuples are $Q_3(T_{\mathsf{drug}}) = \{t_2, t_5\}$ and $|Q_3(T_{\mathsf{drug}})| = 2$. One can directly set the numbers associated with DML, DLQ, and DMLQ to 0 (case 1). Moreover, it is safe to change the number with node M as $3 - 2 = 1$. Similarly, we change the number with L (resp. $\varnothing$) to 1 (resp. 4) (case 2). Consider DL and tuples $Q_3(T_{\mathsf{drug}}) = \{t_2, t_5\}$, it is easy to verify that DL can update t_2 but not t_5, hence the number with DL will be changed as $1 - 1 = 0$ (case 3).

5.2 Closed Rule Sets

A natural question, when searching a lattice, is whether there is any redundancy in the behavior of the rules, so we turn our attention now on how to identify such redundancy.

Closure operator f. Given a lattice $(\mathcal{Q}, \preceq)$ for update Δ and table T, we define a closure operator f. For any $Q \in \mathcal{Q}$, let $f(Q) = \{Q'\}$ and the following properties hold: (1) $Q \preceq Q'$; (2) $|Q(T)| = |Q'(T)|$; and (3) $\nexists Q'' \in \mathcal{Q}$ where $Q' \neq Q''$, $Q' \preceq Q''$, and $|Q'(T)| = |Q''(T)|$.

Intuitively, the closure operator f is to locate the *maximal* ancestor of a query Q that has the same effect on the number tuples they can change. Consider Fig. 2 for example, we have $f(\text{DMLQ}) = \{\text{DL}\}$, and $f(\text{DMQ}) = \{\text{DM, DQ}\}$.

Closed rule sets. Given a lattice $(\mathcal{Q}, \preceq)$, two rules Q and Q' belong to the same *closed rule set*, *iff* $f(Q) = f(Q')$. The smallest (minimal) closed rule set contains one rule Q, *i.e.,* $f(Q) = \{Q\}$ and no other rule Q' where $f(Q') = \{Q\}$.

Example 10: Consider Fig. 2. The shapes identify distinct closed rule sets. For example, the closed rule set for "○" is {DMLQ, DML, DLQ, DL}, since they are connected and have the same affected numbers. Also, the closed rule set for "•" is {DMQ, DM, DQ}, similar for other shapes.

It deserves to note that the concept of closed rule sets is in the instance level, *i.e.,* queries in the same closed rule set will change the same set of tuples for the given dataset. However, they are not the same in the semantic level, *i.e.,* some of them might be valid while the others might be invalid. In order to better understand the above discussion, consider an extreme case that each lattice node can change only one tuple, which makes all candidate queries in one closed rule set. Apparently, they contain both valid and invalid rules. In other words, the closed rule set ignores the factor that whether a rule is valid or not.

Representative rule. One natural question, given a closed rule set, is which query to be verified by the user. The intuition behind our choice is that, the more specific the query is, the easier it is for the user to verify. Hence, we define the most *representative rule* in a closed rule set to be the query Q with the largest number of predicates *w.r.t.* $|\mathsf{attr}(Q)|$.

For instance, in Example 10, the representative rule for the rule set of "○", {DMLQ, DML, DLQ, DL}, is DMLQ.

Benefits of the closed rules set. Any search algorithm over the lattice can benefit from the closed rules set. Given a node in a set, there are consequences that favor the search both if the rule is judged valid or invalid. Remember that we expose and test the representative rule. If it is true, we do not need to compute the updates for any query in the same closed rule set any more. If the answer is no, we also have a benefit in terms of pruning of the nodes, since all the nodes in the set can be safely discharged.

Example 11: Consider Figure 3 and the case that an algorithm has to test node F1. By computing the closed rule set (nodes marked with ○), the rule at the top is tested. If the rule is valid, and therefore being executed, all the nodes marked with ○ will have empty updates now, so we can avoid their computation. But if the rule is invalid, we can prune all the nodes in the set, which is a big benefit compared with the failed test of F1. In the latter case, we would still have to validate the remaining nodes marked with ○, even if we can already derive that they are not valid.

The major difference of our lattice, in contrast to traditional closed item set lattice used for data mining [42], is that our lattice is dynamically changed. More specifically, for each node Q, its associated information $|Q(T)|$ might change during each interaction, such that the closed rule sets will change correspondingly.

6. EXPERIMENTAL STUDY

We implemented FALCON in Java and used PostgreSQL 9.3 as the underlying DBMS. All experiments were conducted on a MacBook Pro with an Intel i7 CPU@2.3Ghz and 16GB of memory. Our frontend extends OpenRefine.

Datasets. We used four real-world datasets and one synthetic dataset, described as follows.

❶ Soccer is a real dataset with 7 attributes and 1625 tuples about soccer players and their clubs scraped from the Web (www.premierleague.com/en-gb.html, www.legaseriea.it/en/, www.bundesliga.com/en/).

❷ Hospital is based on a dataset from US Department of Health & Human Services (http://www.medicare.gov/hospitalcompare/). It has 12 attributes and 100k tuples.

❸ BUS is one of the UK government public datasets available at http://data.gov.uk/data and deals with bus schedules and routes. It contains 15 attributes and 250K tuples.

❹ DBLP is based on the popular collection of authors, publications and venues from http://dblp.uni-trier.de/xml/. We downloaded the whole XML dataset, and translated it into a single relational table with 15 attributes. We considered instances of 1M and 5M tuples, for quality and scalability tests, respectively.

❺ Synth is a dataset we designed starting from the original Soccer dataset in order to study the scalability over the number of tuples and a larger number of attributes. The dataset has 10 attributes and we used a generator from http://www.cs.toronto.edu/tox/toxgene/ to create instances of different sizes.

Algorithms. We implemented several algorithms for the exploration of the lattice. First, we study our own proposals for multi-hop search. Dive is the binary jump algorithm presented in Section 4.2.1. CoDive is its extention to make use of the attributes correlation information, when this is available, as described in Section 4.2.2.

These are compared with one-hop search strategies (Section 4.1). Beside BFS and DFS, we have also implemented Ducc [28], which was designed to reduce the number of tests during the discovery of all the minimal unique column combinations in a given dataset. As we will show in the results, Ducc is better than BFS and DFS for extensive searches of maximal rules in the lattice, but it was not designed to deal with small values of budget B for user interactions.

In addition to these, we also compared our results with the greedy search algorithm for the off-line version of our problem. This algorithm, OffLine, is aware of the valid nodes in the lattice. Given this information, it greedly picks the node that maximizes the error coverage at each step, with the number of steps equals to the budget B.

Baselines. We compared FALCON with four baselines.

❶ *Refine*: Our proposal generalizes the transformation language of existing tools such as OpenRefine (http://openrefine.org/) and Trifacta Wrangler (https://www.trifacta.com/trifacta-wrangler/) [27]. These tools enable users to define transformations by examples exactly as in our setting. Users modify values in a cell for attribute A and the systems suggests possible transformations over the remaining tuples for A. While we do not focus on string manipulation as some of these tools, our language supports rules (*i.e.,* transformations) with look-up over any combination of columns in the relation. In fact, given an update, these tools enable the inference of only two transformations that are comparable to our language: either the single cell is updated (the top of the lattice) or the erroneous value e is replaced with the new value v for all the occurrences in the attribute. The latter corresponds to one of the nodes in our lattice. More precisely, the standardization rule is: UPDATE T SET $A = v$ WHERE $A = e$.

Given this context, a natural baseline algorithm models these transformation tools. This algorithm, namely Refine, checks for every user update the node that generalizes it to a standardization rule, or picks the rule at the top of the lattice if the validation fails.

❷ *Rule-Learning Approaches:* Many previous approaches have concentrated on learning data-quality rules (*e.g.,* [12, 17]). Therefore, we compared our algorithm with one of these methods. More specifically:

(i) starting from a dirty database, we asked users to clean a sample of tuples (part of the budget was used to do this);

(ii) based on the sample tuples, we used a CFD-miner to learn a number of SQL-updates; since it is known that rule-mining algorithms may discover semantically invalid rules (due to *"overfitting"*) we asked users to select a subset of semantically valid rules (the second part of the budget was used for this purpose); and

(iii) we used the set of SQL-updates to repair the dirty instance, and measured the *benefit* score (see below).

❸ *Guided Data Repairs:* To explore the impact of *active learning*, we used GDR [46]. GDR (*"Guided Data Repairs"*) is a recently proposed algorithm that relies on active learning in order to improve the quality of repairs. Given a set of rules, it will incrementally ask users to solicit the right repairs suggested by the rules. We tested an incremental variant of algorithm ❷ above, by using GDR to suggest repairs (*i.e.,* cell updates) to users. In this case, additional budget is used to answer GDR user queries.

❹ *Active Learning in Lattice Traversal:* Finally, we compared our methods to an active learning variant of our lattice-based approach that was designed *ad-hoc* for this purpose. In the active learning algorithm, we first generated some features for each node, including attribute indicator, attribute values, original value, and updated value. We then trained a support vector machine (SVM) model with labeled nodes. Finally we used active learning to select the best node to ask users in each iteration.

Errors and Metrics. Since the considered datasets are clean, we introduce noise to verify the algorithms behaviour in the cleaning process. To start, we manually defined a set of CFDs [16] and fixing rules [43] for each scenario. We used 8 rules for Soccer, 124 rules for Hospital, 8 rules for BUS, 69 rules for DBLP, and 12 rules for Synth.

Afterwards, we used an error-generation tool to inject errors into the clean instances. To make our error-generation more systematic, we relied on an open-source error generation tool by Arocena and others [6]. The tool allows users to inject various kinds of errors within a clean database, both rule-based, random and statistics-based. Being based on an open-source tool, our error generation configuration can be easily shared and reused.

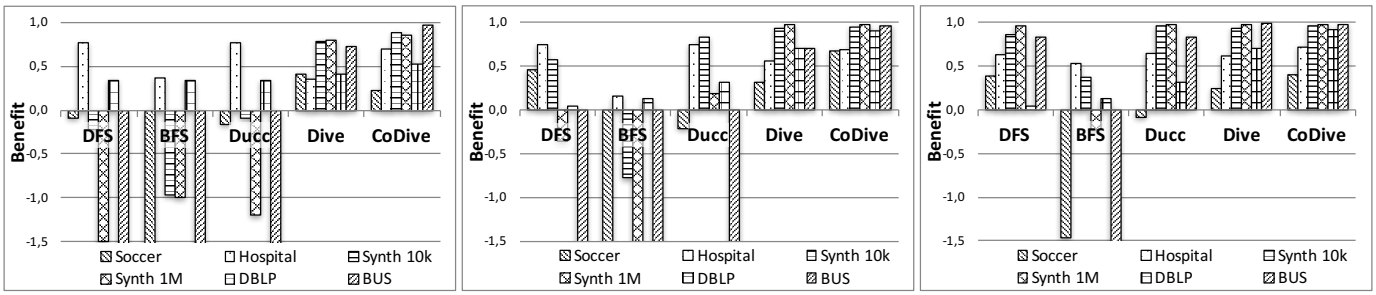
We keep running an algorithm until all the introduced errors are fixed either by a rule or by the user updates. Then, we focus our attention on the *interaction cost*. We adopt natural metrics: the number of user-provided updates U, the number of users' answers for nodes validation A, and we simply add them up to get the total interaction cost T_C. Notice that the latest metric is treating both kinds of interaction with the same weight, *i.e.,* they are considered equally difficult for the user. Despite more sophisticated combinations are possible, we found that the simple sum gives a global overview of the algorithms behaviour that is close to the real overall experience of the users.

In order to have an indicator of the advantage of using interactive cleaning, we also measure the *benefit* of an algorithm in comparison to the manual update of all the errors. We first define the *cost ratio* as the number of actions divided by the number of errors. Manually updating 100 errors requires 100 user actions (updates) for a cost ratio of 1. However, by using our tool, it may be the case that 25 actions can fix 100 errors, therefore the cost ratio would be 0.25. Given an algorithm α, a dataset D, and the interaction cost T_C to obtain a set of queries $\mathcal{Q}$ covering all introduced errors, we define the *benefit* of the algorithm as $\text{BNF}_\alpha = 1 - T_C/|\mathcal{Q}(D)|$.

Finally, we measure the execution times for the algorithms in the generation of the lattice and in its maintenance.

Notice that we do not assume that users always provide correct inputs. On the contrary, the impact of user mistakes is studied in one of our experiments.

Experiments. We conducted five experiments. (i) Exp-1 compares benefits of the various lattice-traversal algorithms with different budget values, and show that CoDive maximizes the benefit. (ii) Exp-2 studies the impact of different



(a) Budget=2 (b) Budget=3 (c) Budget=5

Figure 4: Benefit for the various algorithms for the five datasets.

parameters of the models. In particular, we show that closed rule sets, an optimization technique discussed in Section 5.2, always reduces the cost. (*iii*) Exp-3 compares CoDive with closed rule sets to the four baselines. Interestingly, our algorithm outperforms all of the baselines. (*iv*) Exp-4 studies scalability. (*v*) Finally, Exp-5 investigates the robustness of FALCON *w.r.t.* user mistakes.

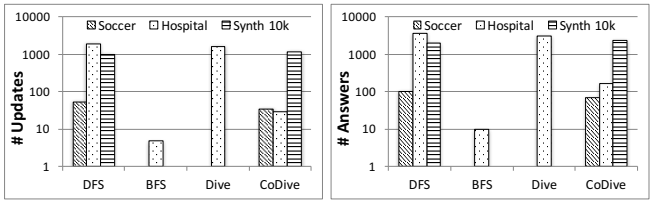
Exp-1: Lattice search algorithms. We now turn our attention to the comparison of the different search algorithms. Figure 4 reports the benefit of each algorithm for the six datasets over increasing budget B (*i.e.,* maximum number of questions after an update).

We start with the setting where the user is willing to answer only two questions (B=2) in Figure 4(a). The proposed algorithms, Dive and CoDive, consistently report a positive gain, which, for CoDive, can be interpreted as a reduction of the total user interaction cost between 22% (Soccer) and 97% (BUS). The plot also reveals that one-hop algorithms fail for the budget exploration of the lattice, with the notable exception of the Hospital dataset. This results is not surprising if we look more closely at this scenario. Hospital schema has a large number of FDs with always one or two attributes in the left hand side (LHS) of the rules. This is reflected in the CFDs that we used to introduce the errors. Rules with one or two LHS attributes are at the bottom of the lattice, and this is the most favourable setting for one-hop based algorithms, since they all start from the bottom. On the other hand, when rules start to have more attributes in the LHS, more nodes must be checked to take a decision, these algorithms fail and Dive and CoDive greatly outperform them. Similar results can be observed with $B = 3$ in Figure 4(b). More details are provided in Appendix D.2.

By increasing the budget to five questions, as reported in Figure 4(c), all algorithms can explore the lattice further at each update and the performance improve accordingly. This improvement is bigger for one-hop based algorithms as they are now able to get closer to the maximal rules in the traversal with a smaller number of updates.

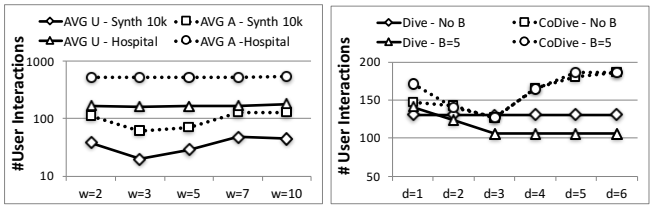
Finally, since algorithm OffLine does not need to perform the search, for each update it is able to identify immediately the maximal rule. Therefore as expected OffLine is always able to completely fix the data with a number of steps that is equal to the number of rules used to introduce noise.

Exp-2: Closed rule sets and parameters. The results for the previous experiments have been conducted with the closed rule sets computed in the lattice. In fact, this optimization enables a reduction both in the number of updates and in the number of questions. To illustrate the impact of the closed rule sets, we executed the lattice search algorithms with and without this optimization, and we measure the difference in the number of required user updates and user answers to cover all errors on three scenarios (Soccer, Hospital and Synth 10k) (see Figure 5). All methods benefit from the optimization, with the exception of Ducc, which does not show any difference, and thus is not reported.



(a) Number of User Updates (b) Number of User Answers

Figure 5: Impact of closed rule sets for $B = 2$.



(a) Avg costs over $B = 2, 3, 5$ for algorithm CoDive wrt w (b) Costs for Synth 1k wrt d for B=5 and without budget

Figure 6: Effects over the interaction cost. U and A are the number of user updates and user answers.

The method that gains most benefit from this optimization is DFS. The explanation is that with low budgets, such as $B = 2$, DFS always reaches the level in the lattice with two attributes. While the rule corresponding to the node may be too general and therefore invalidated by the user, it may be part of a closed rule set. Therefore, the user is offered the representative rule, which is more specific and, in some cases, true. This happens also for rules with only one attribute in the LHS for Hospital, as discussed above. In fact, even BFS, which never goes beyond nodes with only one attribute in the LHS with low B values, benefits from the closed rules set for this dataset.

As shown in Exp-1, on average CoDive has higher benefit values than Dive. However, the quality of CoDive depends on the value for parameter w (Section 4.2.2). We report in Figure 6(a) the experimental results with different w values. Each reported value is the number of user updates U (user answers A) averaged over the results for B equals to 2, 3, and 5. Both for Hospital and Synth 10k the best results are observed with $w = 3$. The parameters does not impact the results for Soccer.

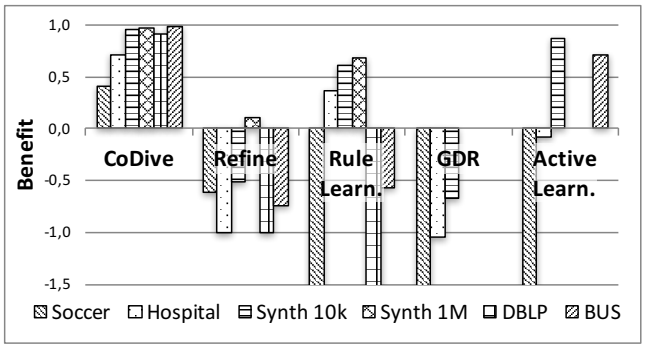


Figure 7: Benefit compared with the baselines.

We also report in Figure 6(b) the experimental results for the synthetic datasets with different values of the parameter d, as introduced for the binary jump algorithm in Section 4.2.1. Experiments over different B values and datasets also confirm that $d = 3$ leads to the best results in terms of optimization of the interaction cost.

Exp-3: Comparison to the baselines. Figure 7 reports a comparison of our CoDive algorithm to the four baselines discussed in Section 6. We fixed a timeout of two hours for all tests. Notice that not all algorithm terminated within the timeout. This accounts for the missing bars in the chart.

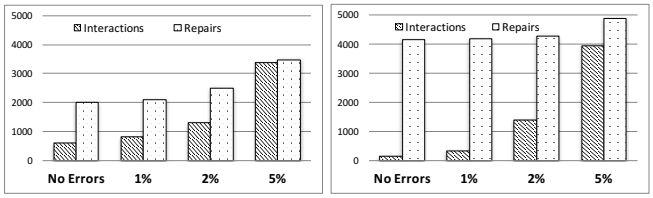
Our approach significantly outperforms all baselines. First, CoDive results are significantly better than those based on rule discovery. This suggests that our novel paradigm for data repairing is an improvement *w.r.t.* previous approaches in which quality rules are established upfront. Interestingly, this is confirmed also in the case in which rule discovery is coupled with an interactive algorithm, like GDR. In fact, the additional number of user interactions needed to run GDR brings to even lower benefit.

Results confirm our intuition that using user updates to lead the discovery of rules in an incremental way yields more complete and effective repairs than state-of-the-art rule-learning algorithms, which can return incomplete or redundant sets of constraints. In fact, in our experiments neither RuleLearning, nor GDR was able to repair all of errors in the data. Detailed comparison is reported in Appendix D.2.

CoDive algorithm also outperforms its active learning variant. Since ActiveLearning shares the same infrastructure as CoDive, here results are better *w.r.t.* RuleLearning and GDR. In fact, as for CoDive, whenever it terminated also ActiveLearning was able to repair all errors. ActiveLearning worked well in datasets with few rules, such as BUS and Synth 10k, while performed poorly in datasets with many rules like Hospital. Appendix C reports further details on active learning algorithm. Overall, however, benefit levels are lower. Hence, the active learning variant pays the price in terms of user-interactions of the additional training phase, which does not bring benefits *w.r.t.* CoDive.

Finally, CoDive outperforms Refine because of the less expressive language in the latter. While we discover rules using any combination of columns, Refine either generates rules for the entire column, which is unlikely to hold for data errors, or rules that update a single tuple. Single tuples updates are always correct and promptly validated, but their very small coverage leads to no benefit in using this tool.

Exp-4: Scalability. We report the performance of the lattice construction and maintenance in Figure 8. Times are reported in *ms* and the y axis is in log scale.



(a) Hospital (b) BUS

Figure 9: Impact of user mistakes.

We start by analyzing the impact of the techniques discussed in Section 5.1 for lattice maintenance. Figure 8(a) shows the total execution time for an update, defined as the time to create the lattice plus the time to update it with rules validated by the user in the interaction. We find it interesting to show that different updates can lead to very different execution times, because of the size of the queries involved in the lattice. Therefore, for the same scenario, we report both the execution times for the first user update, and corresponding interaction, and the times for the 4^{th} user update. For all combinations of updates and scenarios, the incremental maintenance is 3–5 times faster than the naive solution that rebuilds the lattice for every rule validated by the user (4 times faster on average for the first five updates).

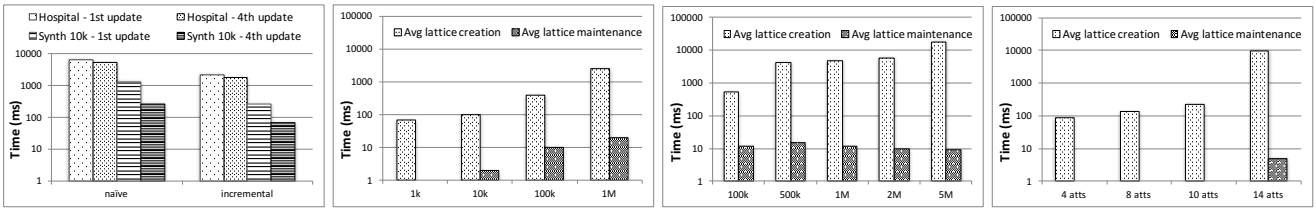
Creating the lattice requires to run queries to collect the data, and intersection over the sets of tuples to find the corresponding number of affected tuples for each node. When the dataset is large, the creation of the lattice can require a couple of seconds, as reported in Figure 8(b-c) for the average of the first ten updates. However, the creation is required only when a new user update is given, and the maintenance of the lattice in the rule validation always requires less than 20ms with our technique.

Finally, we study how the number of attributes in the dataset influences the performance. For this experiment we selected subsets of attributes of Hospital and also extended it with two more attributes by joining another table. Figure 8(d) shows the average times over the first five updates for the creation of the lattice and its maintenance with our technique. While the response time is always below 10ms, the creation of the lattice takes on average about 10 seconds, with a maximum of 30 seconds for the first user update. As discussed in Section 5.1.1, it is important to be able to identify the attributes of interest for the mining to limit the exponential explosion of the number of nodes in the lattice.

Exp-5: User Mistakes. We also tested the robustness of our approach *w.r.t.* to user-errors. That is, we do not assume that users always provide correct answers. On the contrary, assume users may sporadically make mistakes. These may be of two kinds, as follows. We notice that in both cases, our algorithm is essentially self-healing:

(i) The user performs a wrong update. This is the easier case, since we can expect that from a wrong update, only invalid rules are generated; these will be rejected by the user, and the error is fixed.

(ii) Following a valid update, the user wrongfully validates an invalid rule. This case in more delicate, since, following the wrong rule, the algorithm will indeed perform some incorrect updates. Overall, however, this will simply generate more dirtiness in the database, and the user still has a chance to correct this new dirtiness that s/he has introduced in the database in subsequent iterations.



(a) Total execution time (ms) for ith update: lattice creation + maintenance

(b) Avg execution time (ms) with increasing # of tuples for Synth (first 10 updates)

(c) Avg execution time (ms) with increasing # of tuples for DBLP (first 10 updates)

(d) Avg exec. time (ms) with increasing # of attributes for Hospital (first 10 updates)

Figure 8: Efficiency for the lattice creation and maintenance: Dive algorithm, unbounded B.

A key property is that the rules we discover at each step are applied only once – *i.e.,* during the step they were generated in – and therefore they can be fixed by further interaction. This requires that the user is requested to reconsider some previously updated cells. As a consequence, repair ratios decrease in case of errors. In addition, we need to prevent cyclic behaviors. To do this, the system checks updates and notifies users whenever it is updating a cell that has been repaired in previous iterations. This helps users to identify previous mistakes, and prevents cycles.

Figure 9 shows the impact of user mistakes. Assume that users made mistakes with a given probability – ranging from 1% to 5% – and compare results to the case without mistake. Experiments confirm that the system is able to recover from these errors, at the price of more user interactions.

7. RELATED WORK

Data transformation. Interactive systems for data transformation [27,37,44] also reason about the updated attribute to learn transformation rules. They mainly focus on string manipulation and reformatting at the text level. In contrast, we use more expressive SQL scripts. Consequently, we discover not only rules that contain one attribute that is being updated syntactically, but also rules that combine multiple attributes to semantically determine new repairs. Our language and algorithms can lead to smaller interaction cost, as discussed in Section 6 Exp-3.

Machine learning for cleaning. Given a set of user updates, they can be used as training data to train machine learning models, which in turn can be used to predict other repairs [41, 45]. However, ML models are typically *black-boxes* that identify updates without explanations, which are hard to be trusted by users, especially for critical applications that need repairs with guaranteed correctness. Instead, SQLU queries are declarative and are preferred for human validation. Moreover, to train a machine learning model with updates, they must be semantically consistent, *i.e.,* they refer to the same type of errors. In practice, however, this assumption does not always hold since multiple updates may refer to different types of errors. This heterogeneity may hinder the usability of the trained machine learning model for prediction. Different from them, FALCON is bootstrapped by a single update, and ensures the following interactions are related to the queries with consistent semantics.

Query by examples. Several proposals have exploited the opportunity of using examples to discover queries [2,8,9,38, 49,50], schema matchings [35,47], and schema mappings [4]. They mainly focus on finding how to join multiple tables. In contrast to them, we study how to discover SQLU queries on one table, with the main challenge of understanding the update semantics that is not considered by other approaches. From an algorithmic perspective, most of these approaches exploit active learning to validate with users informative examples; we show in Section 6 Exp-3 how other signals, such as correlation, can better guide the search in our setting.

Rule-based data cleaning. Rule-based approaches for data cleaning are divided between methods to discover the rules from clean data [11,12,17,25,40], and algorithms and systems to apply the rules over dirty data to automatically fix the detected errors [7,13,15,18,19,21,23,24,30,32,43,46]. Our proposal overcomes some of the shortcomings in these methods. In terms of rules discovery, mining on dirty data leads to a lot of useless rules, therefore most of the methods report effective results assuming a clean sample. On the contrary, we naturally start from dirty data. In terms of cleaning, we restrict our language to deterministic updates, which do no need variables or placeholders that the users ultimately have to manually verify. In terms of learning from user repairs, the closest approach to our solution is the use of previous repairs to model "repair preferences" [41]. However, this approach needs a set of rules to be given as input and it only refines them, without discovering new ones.

Closed frequent itemset. The concept of closed frequent itemset is widely used in data mining (see [48] for a survey), where it refers to a set of itemsets that are both frequent (*i.e.,* the support value is above a given threshold) and closed (*i.e.,* there is no superset that is closed). In fact, our closed rule set is inspired from closed frequent itemset, with the major difference that our data structure (*i.e.,* the lattice) keeps changing during interactions. Traditionally, the search space for closed frequent itemset in data ming is static.

8. CONCLUSION AND FUTURE WORK

We have presented FALCON, an interactive, declarative and deterministic data cleaning system. We have demonstrated that FALCON can effectively interact with users to generalize user-solicited updates, and clean-up data with a significant benefit *w.r.t.* the number of required interactions.

A number of possible future studies using FALCON are apparent. First of all, we plan to extend it by using external sources, as remarked in Appendix B. Moreover, we will leverage the information obtained from previous interactions with the user *w.r.t.* multiple data updates.

Acknowledgement. This work was partly supported by the 973 Program of China (2015CB358700), NSF of China (61422205, 61472198), Huawei, Shenzhou, Tencent, FDCT/116/2013/A3, MYRG105(Y1-L3)-FST13-GZ, National High-Tech R&D (863) Program of China (2012AA012600), and the Chinese Special Project of Science and Technology (2013zx01039-002-002).

9. REFERENCES

[1] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[2] A. Abouzied, J. M. Hellerstein, and A. Silberschatz. Playful query specification with dataplay. *PVLDB*, 5(12), 2012.

[3] S. Albers. Online algorithms: a survey. *Math. Program.*, 97(1-2), 2003.

[4] B. Alexe, L. Chiticariu, R. J. Miller, and W. C. Tan. Muse: Mapping understanding and design by example. In *ICDE*, pages 10–19, 2008.

[5] C. Ambühl. Offline list update is np-hard. In *Algorithms - ESA 2000, 8th Annual European Symposium*, 2000.

[6] P. C. Arocena, B. Glavic, G. Mecca, R. J. Miller, P. Papotti, and D. Santoro. Messing up with BART: error generation for evaluating data-cleaning algorithms. *PVLDB*, 9(2), 2015.

[7] P. Bohannon, M. Flaster, W. Fan, and R. Rastogi. A cost-based model and effective heuristic for repairing constraints by value modification. In *SIGMOD*, 2005.

[8] A. Bonifati, R. Ciucanu, and S. Staworko. Interactive inference of join queries. In *EDBT*, 2014.

[9] A. Bonifati, R. Ciucanu, and S. Staworko. Interactive join query inference with JIM. *PVLDB*, 7(13), 2014.

[10] C. Chang and C. Lin. LIBSVM: A library for support vector machines. *ACMTIST*, 2(3):27, 2011.

[11] F. Chiang and R. J. Miller. Discovering data quality rules. *PVLDB*, 1(1), 2008.

[12] X. Chu, I. F. Ilyas, and P. Papotti. Discovering denial constraints. *PVLDB*, 6(13), 2013.

[13] M. Dallachiesa, A. Ebaid, A. Eldawy, A. K. Elmagarmid, I. F. Ilyas, M. Ouzzani, and N. Tang. NADEEF: a commodity data cleaning system. In *SIGMOD*, 2013.

[14] O. Deshpande, D. S. Lamba, M. Tourn, S. Das, S. Subramaniam, A. Rajaraman, V. Harinarayan, and A. Doan. Building, maintaining, and using knowledge bases: A report from the trenches. In *SIGMOD*, 2013.

[15] A. Ebaid, A. K. Elmagarmid, I. F. Ilyas, M. Ouzzani, J. Quiané-Ruiz, N. Tang, and S. Yin. NADEEF: A generalized data cleaning system. *PVLDB*, 6(12):1218–1221, 2013.

[16] W. Fan, F. Geerts, X. Jia, and A. Kementsietsidis. Conditional functional dependencies for capturing data inconsistencies. *ACM Trans. Database Syst.*, 33(2), 2008.

[17] W. Fan, F. Geerts, J. Li, and M. Xiong. Discovering conditional functional dependencies. *IEEE Trans. Knowl. Data Eng.*, 23(5), 2011.

[18] W. Fan, F. Geerts, N. Tang, and W. Yu. Inferring data currency and consistency for conflict resolution. In *ICDE*, 2013.

[19] W. Fan, J. Li, S. Ma, N. Tang, and W. Yu. Interaction between record matching and data repairing. In *SIGMOD*, 2011.

[20] W. Fan, J. Li, S. Ma, N. Tang, and W. Yu. Towards certain fixes with editing rules and master data. *VLDB J.*, 21(2), 2012.

[21] H. Galhardas, D. Florescu, D. Shasha, E. Simon, and C.-A. Saita. Declarative data cleaning: Language, model, and algorithms. In *VLDB*, 2001.

[22] F. Geerts, G. Mecca, P. Papotti, and D. Santoro. The LLUNATIC data-cleaning framework. *PVLDB*, 6(9), 2013.

[23] F. Geerts, G. Mecca, P. Papotti, and D. Santoro. Mapping and Cleaning. In *ICDE*, pages 232–243, 2014.

[24] F. Geerts, G. Mecca, P. Papotti, and D. Santoro. That's all folks! LLUNATIC goes open source. *PVLDB*, 7(13):1565–1568, 2014.

[25] L. Golab, H. J. Karloff, F. Korn, B. Saha, and D. Srivastava. Discovering conservation rules. In *ICDE*, 2012.

[26] A. Y. Halevy. Answering queries using views: A survey. *VLDB J.*, 10(4), 2001.

[27] J. Heer, J. M. Hellerstein, and S. Kandel. Predictive interaction for data transformation. In *CIDR*, 2015.

[28] A. Heise, J. Quiané-Ruiz, Z. Abedjan, A. Jentzsch, and F. Naumann. Scalable discovery of unique column combinations. *PVLDB*, 7(4), 2013.

[29] I. F. Ilyas, V. Markl, P. J. Haas, P. Brown, and A. Aboulnaga. CORDS: automatic discovery of correlations and soft functional dependencies. In *SIGMOD*, 2004.

[30] M. Interlandi and N. Tang. Proof positive and negative in data cleaning. In *ICDE*, 2015.

[31] S. Kandel, A. Paepcke, J. M. Hellerstein, and J. Heer. Enterprise data analysis and visualization: An interview study. *IEEE Trans. Vis. Comput. Graph.*, 18(12), 2012.

[32] Z. Khayyat, I. F. Ilyas, A. Jindal, S. Madden, M. Ouzzani, J.-A. Quiane-Ruiz, P. Papotti, N. Tang, and S. Yin. BigDansing: a system for big data cleansing. In *SIGMOD*, 2015.

[33] J. Manyika. Big data: The next frontier for innovation, competition, and productivity. Technical report, McKinsey Global Institute, 2011.

[34] C. H. Papadimitriou. *Computational Complexity*. Addison Wesley, 1994.

[35] L. Qian, M. J. Cafarella, and H. V. Jagadish. Sample-driven schema mapping. In *SIGMOD*, 2012.

[36] G. Ramalingam and T. W. Reps. A categorized bibliography on incremental computation. In *POPL*, 1993.

[37] V. Raman and J. M. Hellerstein. Potter's wheel: An interactive data cleaning system. In *VLDB*, 2001.

[38] Y. Shen, K. Chakrabarti, S. Chaudhuri, B. Ding, and L. Novik. Discovering queries based on example tuples. In *SIGMOD*, pages 493–504, 2014.

[39] D. D. Sleator and R. E. Tarjan. Amortized efficiency of list update and paging rules. *Commun. ACM*, 28(2), 1985.

[40] S. Song and L. Chen. Efficient discovery of similarity constraints for matching dependencies. *Data Knowl. Eng.*, 87, 2013.

[41] M. Volkovs, F. Chiang, J. Szlichta, and R. J. Miller. Continuous data cleaning. In *ICDE*, 2014.

[42] J. Wang, J. Han, and J. Pei. CLOSET+: searching for the best strategies for mining frequent closed itemsets. In *SIGKDD*, 2003.

[43] J. Wang and N. Tang. Towards dependable data repairing with fixing rules. In *SIGMOD*, 2014.

[44] B. Wu and C. A. Knoblock. An iterative approach to synthesize data transformation programs. In *IJCAI*, pages 1726–1732, 2015.

[45] M. Yakout, L. Berti-Equille, and A. K. Elmagarmid. Don't be scared: use scalable automatic repairing with maximal likelihood and bounded changes. In *SIGMOD*, 2013.

[46] M. Yakout, A. K. Elmagarmid, J. Neville, M. Ouzzani, and I. F. Ilyas. Guided data repair. *PVLDB*, 2011.

[47] Z. Yan, N. Zheng, Z. G. Ives, P. P. Talukdar, and C. Yu. Actively soliciting feedback for query answers in keyword search-based data integration. *PVLDB*, 6(3):205–216, 2013.

[48] M. J. Zaki and W. Meira. *Data Mining and Analysis: Fundamental Concepts and Algorithms*. Cambridge University Press, 2014.

[49] M. Zhang, H. Elmeleegy, C. M. Procopiuc, and D. Srivastava. Reverse engineering complex join queries. In *SIGMOD*, 2013.

[50] M. M. Zloof. Query by example. In *AFIPS*. ACM, 1975.

APPENDIX

A. SUMMARY OF NOTATION

We summarize the notations used in the paper in Table 3.

B. USING EXTERNAL SOURCES

Symbol	Description
Q	a SQLU query, or a data quality rule
$Q(T)$	affected tuples of Q over table T
$Q_1 \preceq Q_2$	Q_1 is contained by Q_2
$\mathsf{attr}(Q)$	attributes in the WHERE condition of Q
Δ: $t[A] \leftarrow a'$	an update of $t[A]$ to a'
$(\mathcal{Q}, \preceq)$ *w.r.t.* Δ	a lattice of queries $\mathcal{Q}$ on partial order $\preceq$
Q^{Υ} ($\mathcal{Q}^{\Upsilon}$)	the set of queries that Q (queries $\mathcal{Q}$) contains
Q_{Λ} ($\mathcal{Q}_{\Lambda}$)	the set of queries that contains Q (queries $\mathcal{Q}$)

Table 3: Notations used in the paper.

Indicators				AttributeValues				Original	Updated
D	M	L	Q	D	M	L	Q	M	M
1	2	1	0	11 Nov	statin	Austin	null	statin	$C_{22}H_{28}F$

Table 4: Features of node DML.

In this section, we discuss an extension of our system when external sources are present. Often times, external sources (*e.g.,* master data) are available and contain high quality data. Next, we shall discuss how to leverage such information by FALCON.

Consider a dirty table T with schema R and master data M with schema R_m. Assume without loss of generality that $|R| = |R_m|$, and the alignment of attribute from each attribute $A \in R$ to $A \in R'$ is given. For all other attributes in either relation that are not aligned will be ignored. Moreover, given the update $\Delta : t[A] \leftarrow a'$, we assume that $A \in R$.

UPDATE T SET $T.A = M.A'$

FROM T, M WHERE $T[X] = M[X']$

Note that, differently from the SQLU queries defined in Section 2.1, we enforce the condition that $A \notin X$. The reason is that we assume that master data contains only correct values, but not errors. In such case, the number of potential queries is $2^{|R|-1}$, which is the number of all combinations of attributes in $R \backslash \{A\}$.

From the extension, we can repair errors from instance level to schema level by using the same lattice and the same algorithms.

C. ACTIVE LEARNING APPROACH

Active learning is a special case of semi-supervised machine learning with the goal of substantially reducing the number of labelling when training a model. All the labels for learning are obtained without reference to the learning algorithm, while in active learning the learner interactively chooses which data points to label. The hope of active learning is that interaction can substantially reduce the number of labels required. The method relies on interactively querying the user for labelling the data trying to maximize the benefit for the actual learning algorithm.

We adopt a similar idea in our setting, as described below.

In order to use active learning in our problem, *i.e.,* to predict that which node (or query) in the lattice is valid or invalid, we face two main issues to be addressed.

(1) We need to generate features for nodes in the lattice, which are used to capture their characteristics.

(2) We need to use these features to select the node with the maximum benefit to be labelled, which is then interact with users to verify the selected node.

Rank	Attributes Set	Correlation
1	{Stadium, Club Country}	1
2	{Soccer Manager, Soccer Club }	1
3	{Stadium, Soccer Club}	0.822
4	{Stadium, Soccer Manager}	0.789
5	{Stadium, Soccer Club, Soccer Manager}	0.654
...	...	...
99	{Stadium, Playercountry, Soccer Club}	0.303
100	{Stadium, Position}	0.006

Table 5: Correlation of attributes in Soccer dataset when Stadium is updated.

We explain in more detail about the implementation of the above two steps below.

Feature Selection. We generate a number of features for each node Q and train a Support Vector Machine (SVM) model with LIBSVM [10]. For a node Q, the features include attribute indicator, attribute value, the original value before the update, and the updated value. Attribute indicator indicates whether the attribute is included in the node (rule): if included, the indicator is 1; 0 otherwise. While if the attribute is being updated, the indicator value is 2.

Question Generation. There are two phases in question generation. First, in the initial 20 user updates, we use Ducc to explore the lattice to label the nodes, taking the nodes (and the corresponding features) from the user labelling as the training data to train a SVM model that bootstraps the active learning. Second, in each iteration, we apply the SVM model to predict the label and corresponding probability of each node, and select the node with the highest probability of being valid (reported by SVM) to ask users. After obtaining a label from user, we use **lattice pruning** technique (discussed in Section 3) to label other nodes in the lattice and add them to existing training data to re-train the SVM model.

We illustrate by an example for the active learning method.

Example 12: Consider node DML in Figure 2. Firstly, we generate features as illustrated in Table 4. Attribute indicator of D is 1 because node DML includes attribute D, indicator of M is 2 since it is the updated attribute. Attribute values are the corresponding values taken from the update Δ_3 in Example 1. The original value and updated value for the update are shown in the table.

Secondly, in each iteration, we apply SVM model to predict the probability of being valid of each node in the lattice. Suppose node ML has the highest probability to be valid that is 0.78. We then ask the user to label node ML and, since in our example the node is valid, we label all nodes above ML, *i.e.,* {ML, DML, MLQ, DMLQ} to be valid and we add them to re-train the SVM model.

D. ADDITIONAL EXPERIMENTS

D.1 Correlation Score Results

Correlation guides the search to nodes that are likely to have a semantic connection. How to compute correlation is discussed in Section 4.2.2 to improve the binary jump strategy. This is crucial in order to discover set of attributes that form rules that are worth validating with the user. Note

	Soccer		Hospital		Synth 10k		Synth 1M		DBLP		BUS	
	U	A	U	A	U	A	U	A	U	A	U	A
DFS	11	33	**129**	**387**	177	531	5094	15282	1462	4386	3646	10938
BFS	82	246	423	1269	729	2187	14035	42105	1338	4014	4172	12516
Ducc	25	75	**129**	**387**	70	210	3083	9249	1122	3036	3646	10938
Dive	15	41	219	657	29	87	**74**	**222**	462	1386	312	936
CoDive	**8**	**19**	206	412	**24**	**72**	**74**	**222**	**140**	**420**	**48**	**144**
$\|Q(T)\|$	82		2000		1640		15000		6086		4172	

Table 6: Comparison of the lattice search algorithms with $B = 3$: U is the number of user updates, A is the number of user answers, and $|Q(T)|$ is the total number of errors.

	Soccer		Hospital		Synth 10k		Synth 1M		DBLP		BUS	
	T_C	Rep	T_C	Rep	T_C	Rep	T_C	Rep	T_C	Rep	T_C	Rep
CoDive B=5	**49**	82	**567**	2000	**70**	1640	**394**	15000	**560**	6086	**96**	4172
Refine	132	82	4000	2000	2470	1640	13326	15000	12172	6086	7258	4172
Rule Learning	194	27	315	500	474	1212	4800	15000	502	0	1191	757
GDR	225	30	1025	500	1578	943	-	-	-	-	-	-
Active Learning	217	82	2157	2000	214	1640	-	-	-	-	1220	4172
$\|Q(T)\|$	82		2000		1640		15000		6086		4172	

Table 7: Comparison of the baselines. Here T_C is the total interaction cost for the user, Rep is the number of repaired cells, and $|Q(T)|$ is the number of errors.

that if many null values are present, we only count non-null values, and the attributes with many null values will have low correlation score. To clarify the role of the correlation score, consider the following example.

Δ_1: t_1[Stadium] ← "Volkswagen Arena" (from "Weserstadion")

The user updates attribute Stadium. Table 5 shows a summary of the correlation scores, relative to attribute Stadium, for attributes in the Soccer dataset. Every set of attributes in the table can be seen as the left hand side attributes to form a FD. In fact, we know the conclusion of the rule for the given update (the Stadium attribute), but we do not know which attributes to use in the premise of the rule. From the correlation scores, we can deduce that each Stadium usually belongs to one Soccer Club and has one Soccer Manager, while each Stadium could have many Positions. Thus the correlation score of Stadium, SoccerClub, SoccerManager (row with rank 5) is much larger than that of Stadium and Position (row at rank 100). Our algorithm uses this intuition to guide the search strategy, and the experiment results also verify that the correlations are effective in avoiding rules that are unlikely to be validated by the user, such as the one deriving from row at rank 100.

D.2 More Details on Exp-1 and Exp-3

We now discuss in more details the different search algorithms and baselines for all datasets.

As reported in Table 6, all search algorithms, with the exception of BFS, lead to a clean dataset with a number of user updates U that is smaller than the number of errors in the data ($|Q(T)|$, reported at the bottom). When considering the user answers (A), their number is from 4 to 68 times smaller than the cost of manually fixing the errors (without any rule nor tool), when considering the best performing algorithm (numbers in bold). In particular, both for number of required updates and for number of required answers, CoDive is always the method with the lowest effort, with the exception of the Hospital dataset. In this case, DFS and Ducc perform better because of the simple rules that have been used to model the injection of the errors in the data. All rules for this dataset have only one or two attributes in the left hand side of the rules, such as Zip → State or Address, City → State. In these cases, the correct rules are at the bottom of the lattice, which is the level that DFS and Ducc explore first. On the contrary, if our algorithms miss the correct node at the bottom, they would start exploring the rest of the lattice and converge to the bottom again slowly, thus with a larger number of questions. Notice that, as discussed in Exp-2, the closed rule sets optimization shows a significant improvement on the Hospital scenario for the DFS algorithm.

We also remark that the Hospital dataset has a number of rather specific features. This dataset was created by joining several tables, originally in normal form, in order to obtain a large number of functional dependencies and redundancy in the data for testing rule-based data repair algorithms [20,22]. Given the highly denormalized table resulting from these joins, the dataset should not be considered representative of a standard data cleaning task.

Table 7 reports the results for CoDive compared with the baselines. Missing numbers denote cases for which the tool was stopped after the fixed timeout (two hours) for all tests. We observe that Rule Learning and GDR were not able to cover all errors because of the limited scope of the discovered rules. This is due to the limited size of the sample used in mining. A larger sample would lead to better results in terms of recall, but with a higher cost for the collection of the clean tuples. Refine is always able to detect all errors, but with a much larger number of interaction because of its less expressive language, compared to CoDive. Finally, active learning has worse performance *w.r.t.* CoDive because of required training data, and in two cases with large dataset it was not able to terminate before the timeout.